Lesson 6: Building a Performance Culture

Rico Mariani

Architect, CLR Performance Team

http://blogs.msdn.com/ricom/

Rule #1

- Measure
  - Just thinking about what to measure will help you do a good job
  - If you don't measure you can be sure it will be slow, big, or whatever else you don't want
  - If you haven't measured, your job's not finished

Rule #2

- Do Your Homework
  - Good engineering requires you to understand your raw materials
  - What are the key properties of your framework? Your processor? Your target system?

No More Rules

- There are very few absolutes in the performance biz
- Performance work is plagued with powerful secondary and tertiary effects that often dwarf what we think are the primary effects
- Whenever considering advice, you must remember Rule #1

Performance Culture

- **Budget**
  - Budgeting is an exercise to assess the value of a new feature and the cost your customer will bear
- **Plan**
  - Validate your design against the budget, this is a risk assessment
- **Verify**
  - Measure the final results, discard failures without remorse or penalty—don't make us live with them

Performance Processes

- Teams that have good performance culture use common patterns in their process
  - Well-defined benchmarks, run frequently
  - Rapid identification and tracking of regressions
  - End-to-end performance goals based on customer needs, with tracking
  - Broad unit testing before and after check-in
  - Long-term goals staged in short-term ways—that are achievable to yield steady progress

Simple Design: Usable First, Then Re-Usable

- When everything is polymorphic and points to everything else in a fully general way, you have a disaster of over-design, not a gem of generality
- Focus on your customers' needs and you will be much safer

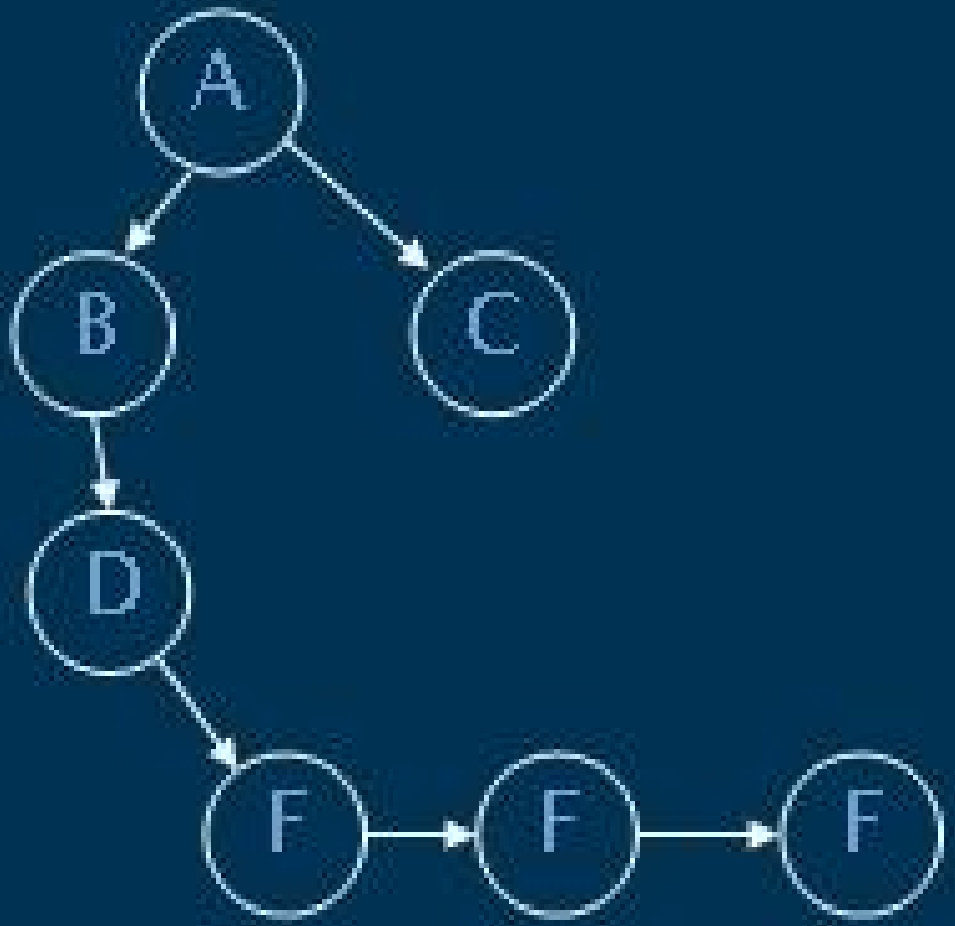
The Root of All Evil

- "Premature Optimization" vs. "Performance Planning"
- There are many ways to be sloppy and as many excuses
- **Engineering is a quantitative discipline**
- Don't let nifty-sounding slogans stand between you and greatness

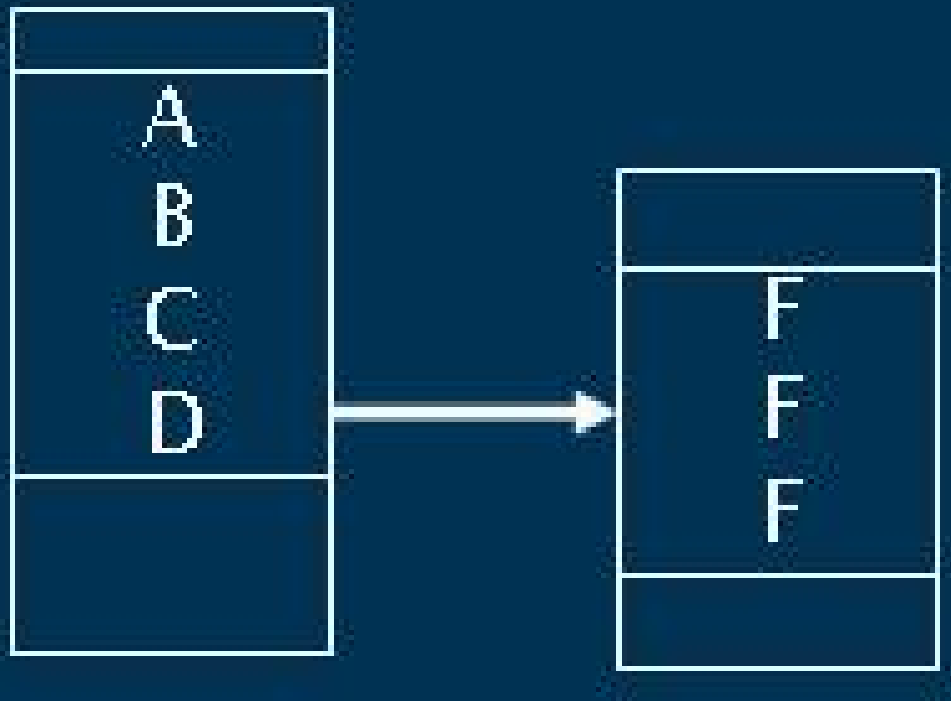
Leveraging Modern Processors: Locality Is Everything

- When I say locality is everything, what I mean of course is that you must apply Rule #1, as I never say anything absolute
- Now, imagine your memory is sliced into pieces… because it is

Less of This



More of This



Some Anecdotes
It's Easy to Accidentally Go Wrong

- Message handling creating enumeration objects in nested loops
- Multiple levels of indirection to support growth of data structures that rarely grow
- Hash and comparison functions that call string splitting functions

Common Themes #1
Reducing Working Set

- Working Set refers to the number of pages of virtual memory committed to a given process, usually dominated by Modules
- Estimate the size of the code you expect to add to support a new feature, track before you check in your code, and then in builds after
- Pay special attention to code that runs even when your new feature isn't being used, costs that are not "pay for play" are to be avoided
- Plan for the cost of accessed data in the module as well as code
- Consolidate new features with old rather than (partly) duplicating code—sounds obvious but we often fail to do it
- Write a unit test to verify that the cost is what you think it is, plan to submit this to your perf team as soon as is reasonable
- Choose a space budget well before its time to check in the code —last-minute justification is a sign of absent planning

Common Themes #2
Reduce Private Pages

- Private memory is defined as memory allocated for a process which cannot be shared by other processes
- Unmanaged modules usually are about 5% private, managed modules are closer to 15–20% private, the rest comes from dynamic data
- Estimate your total memory heap usage, consider the peak levels during initialization and steady state
- When adding features, consider "collateral damage" done to existing classes/structures, estimate growth in these existing structures
- Use heap analysis tools like "CLR Profiler" to get an idea of how many objects of types you expect to affect are typically in memory in your scenarios
- Highlight any non-pay-for-play costs that are incurred on the heap

Common Themes #3
Reduce Startup Time

- Startup refers to the CPU time to get your application to responsive UI
- Cost Driver #1—Soft Faults:
  - A requested disk I/O that could be satisfied from the disk cache
  - A part of a module that is already loaded in another process was needed in this process
  - A page of zeroed memory was needed in this process
  - Charge yourself about 1us (swag) for each 4k of the above
- Cost Driver #2—Actual Disk I/O:
  - Significantly (example: 1000x) more expensive than the above
  - A part of a module that was not already loaded elsewhere was needed in this process (hard fault, unshared pages more expensive than shared)
  - A piece of swapped out read/write data was needed
  - An un-cached piece of a file was referenced (normal disk I/O)
  - Charge yourself about 1ms (swag) for each 4k of the above
- Cost of running the code is often dwarfed by the cost of bringing it into the process
- Give special attention to the cost of reading your initialization files/state, including registry

Final Words

- Understand your goals
- Understand the costs of what you use
- Be ruthless about measuring so that you've done the full job
- Keep reading and experimenting so you can learn aspects of the system that are most relevant to you
- Share your wisdom with your friends
- Insist on performance culture in your group
- Don't forget Rule #1 and Rule #2 !!!

Resources

- http://blogs.msdn.com/ricom/
- Improving Microsoft® .NET Application Performance and Scalability
  http://msdn.microsoft.com/perf/
- http://devdiv/clrperf/

Questions?